AFRL-OSR-VA-TR-2014-0194

AN INTEGRATED ARCHITECTURE FOR AUTOMATIC INDICATION, AVOIDANCE

Dongyan Xu
PURDUE UNIVERSITY

08/20/2014
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ RTC
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 06/08/2014	**2. REPORT TYPE** Final Performance Report	**3. DATES COVERED** *(From - To)* 4/1/2010 – 3/31/2014

4. TITLE AND SUBTITLE An Integrated Architecture for Automatic Indication, Avoidance and Profiling of Kernel Rootkit Attacks	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA9550-10-1-0099
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Xu, Dongyan; Spafford, Eugene H.; Jiang, Xuxian	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Purdue University, 401 S. Grant St., W. Lafayette, IN 47907; North Carolina State University, 890 Oval Dr., Raleigh, NC 27695	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Office of Scientific Research, 875 N. Randolph St., RM3112, Arlington, VA 22203	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFOSR
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The objective of this project is to mitigate or eliminate threats of kernel rootkits against production computer systems. The main goal of this research is the development of an integrated, virtualization-based architecture for automatic *indication, avoidance* and *profiling* of kernel rootkit attacks while maintaining non-stop production system operation. Under this architecture, a production system (running as a virtual machine or VM) executes at full speed under normal circumstances, while the proposed architecture watches out for the first sign of a kernel rootkit attack and indicates the attack right before it strikes. In response, the production VM "splits" into two copies: one is the same production VM running uninterrupted and without the negative impact of the rootkit; while the other one is a live profiling VM which will generate a multi-aspect profile of the kernel rootkit. Moreover, the profile will guide the generation of a variety of kernel attack defense techniques, which will be applied back to the production system and shield it from future rootkit attacks.

15. SUBJECT TERMS

Operating System Security, Virtualization Technology, Malware Defense

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dongyan Xu
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		16	**19b. TELEPHONE NUMBER** *(include area code)* 765-494-6182

AFOSR Final Performance Report

Project Title:	An Integrated Architecture for Automatic Indication, Avoidance, and Profiling of Kernel Rootkit Attacks
Award Number:	FA9550-10-1-0099
Duration:	April 1, 2010 – March 31, 2014

Program Manager: Dr. Robert Herklotz
Information Operations and Security
Air Force Office of Scientific Research
875 N. Randolph Street
Arlington, VA 22203
Phone: (703) 696-6565

PI: Prof. Dongyan Xu
Department of Computer Science and CERIAS
Purdue University
West Lafayette, IN 47907
Phone: (765) 494-6182
Email: dxu@purdue.edu

Co-PI: Prof. Eugene H. Spafford
Department of Computer Science and CERIAS
Purdue University
West Lafayette, IN 47907
Phone: (765) 494-7825
Email: spaf@purdue.edu

Subcontract PI: Prof. Xuxian Jiang
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
Phone: (919) 513-7835
Email: jiang@cs.ncsu.edu

1. Project Summary

Objective The objective of this project is to mitigate or eliminate threats of kernel rootkits against production computer systems. As one of the most elusive types of malware, kernel rootkits are designed to stealthily subvert the operating system kernel – the software root of trust of a computer system. With their omnipotence inside the compromised systems, kernel rootkits have increasingly been used to assist attackers in a variety of malicious activities, such as opening system backdoors, stealing private data, escalating attacker process privileges, and tampering with anti-malware facilities. Unfortunately, the state-of-the-art in kernel rootkit defense is mainly reactive and is in a fundamentally disadvantageous position relative to the kernel attacks. The three main research tasks proposed in this project aim to make a difference by grabbing the "upper hand" in the arms race against kernel rootkits.

Approach The cornerstone of this research is the development of an integrated, virtualization-based architecture for automatic *indication*, *avoidance* and *profiling* of kernel rootkit attacks while maintaining non-stop production system operation. Under this architecture, a production system (running as a virtual machine or VM) executes at full speed under normal circumstances, while the proposed architecture watches out for the first sign of a kernel rootkit attack and indicates the attack right before it strikes. In response, the production VM "splits" into two copies: one is the same production VM running uninterrupted and without the negative impact of the rootkit; while the other one is a live profiling VM which will generate a multi-aspect profile of the kernel rootkit. Moreover, the profile will guide the generation of a variety of kernel attack defense techniques, which will be applied back to the production system and shield it from future rootkit attacks.

2. Accomplishments

The major accomplishments of this project are highlighted as follows.

- This research has resulted in a number of new scientific concepts and results in kernel malware detection, prevention, and profiling (details in Section 3). The research efforts have helped establish kernel malware defense as a major area in computer systems security research. The techniques produced by this research, such as kernel memory shadowing, kernel data mapping and access profiling, process implanting, and process out-grafting are frequently cited by subsequent literatures.

- In addition to scientific concepts and results, this research has generated a number of open-source software prototypes (e.g., NICKLE-KVM, DataGene, Timescope, Process Implanting, Process Out-grafting, and FACE-CHANGE). They are available for distribution to researchers and developers. In particular, NICKLE-KVM and DataGene were adopted as the basis for developing a kernel-level intrusion detection system, which was part of an Army CERDEC-funded project "Host-Centric Intrusion Detection and Reaction" (Contract W15P7T-11-C-A021, 06/2011-05/2013, Main performer: Applied Communication Sciences). Process Implanting and FACE-CHANGE have been transferred (as potential code base) to Intelligent Automation, Inc. for an upcoming AFRL-funded project “Versatile Live Patching System”.

- This research has involved a number of graduate students, who obtained valuable scientific training and hands-on experience in kernel malware defense and virtualization technology. In particular, five students have earned their PhD degrees and started pursuing their research careers in academia and industry: Dr. Dannie Stanley (PhD 2013, assistant professor at Taylor University), Dr. Zhi Wang (PhD 2012, assistant professor at Florida State University), Dr. Deepa Srinivasan (PhD 2012, senior software engineer at Microsoft), Dr. Zhiqiang Lin (PhD 2011, assistant professor at University of Texas at Dallas and **2014 AFOSR Young Investigator**), and Dr. Junghwan Rhee (PhD 2011, researcher at NEC Labs America).

3. Research Activities and New Findings

The following subsections will present more technical details about the major research activities and their scientific findings in this project. Many of these results have been published in peer-reviewed conferences and journal. A complete listing of publications can be found in Section 4.

3.1 Kernel Rootkit Prevention with Performance Optimization

The first task of this research is to explore advanced kernel rootkit prevention/avoidance techniques that are more efficient and effective than the state of the art, including the PIs’ earlier work called NICKLE. Solutions like NICKLE have been created to prevent kernel rootkits by relocating the vulnerable physical system to a guest VM and enforcing a $W \oplus KX$ memory access control policy from the host virtual machine monitor (VMM or hypervisor). The $W \oplus KX$ memory access control policy guarantees that no region of guest memory is both writable *and* kernel-executable.

In this research, it is observed that the guest system must have a way to bypass the $W \oplus KX$ restriction to load *valid* kernel code, such as kernel drivers, into memory. To distinguish between valid kernel code and malicious kernel rootkit code, NICKLE and others use cryptographic hashes for verification. Offline, a cryptographic hash is calculated for each piece of valid code that may get loaded into the guest kernel. Online, the VMM intercepts each guest attempt to load new kernel code and calculates a hash for the code. If the online hash matches an offline hash, the load is allowed.

It is also observed that some modern kernels however, are “self-patching;” they may patch kernel code at run-time. If the patch is applied *prior* to hash verification, then the hashes will not match. If the patch is applied *after* hash verification, then the memory will be read-only, due to $W \oplus KX$ enforcement, and the patch will fail. Such run-time patching may occur for a variety of reason including: CPU optimizations, multiprocessor compatibility adjustments, and advanced debugging. The previous hash verification procedure (including that introduced by NICKLE) cannot handle such modifications.

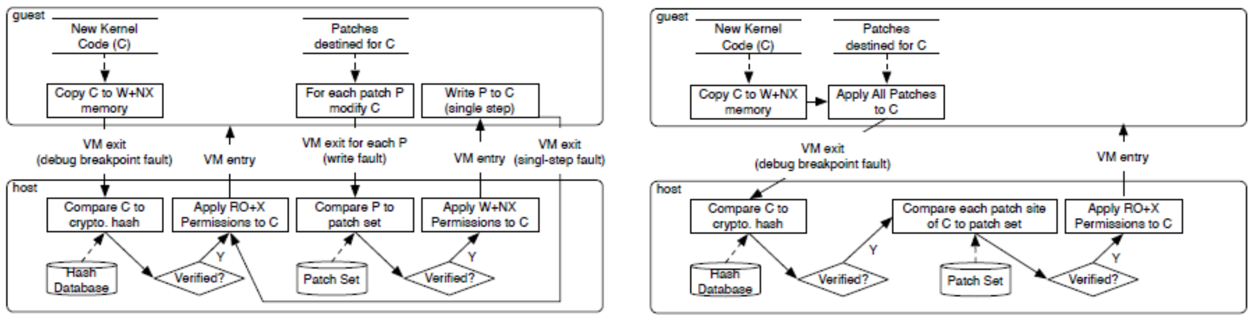

guest
New Kernel Code (C)
Patches destined for C
Copy C to W+NX memory
For each patch P modify C
Write P to C (single step)
VM exit (debug breakpoint fault)
VM entry
VM exit for each P (write fault)
VM entry
VM exit (singl-step fault)
host
Compare C to crypto. hash
Apply RO+X Permissions to C
Compare P to patch set
Apply W+NX Permissions to C
Hash Database
Verified?
Y
Patch Set
Verified?
Y
guest
New Kernel Code (C)
Patches destined for C
Copy C to W+NX memory
Apply All Patches to C
VM exit (debug breakpoint fault)
VM entry
host
Compare C to crypto. hash
Compare each patch site of C to patch set
Apply RO+X Permissions to C
Hash Database
Verified?
Y
Patch Set
Verified?
Y

Figure 1: Runtime (left) and load time (right) verification of self-patching guest kernel code

To address these challenges, efforts in this project have resulted in a new system that verifies the integrity of each instruction introduced by a self-patching kernel. Each instruction is verified by comparing it to a whitelist of valid instruction patches (Figure 1). The whitelist is generated ahead of time while the guest is offline. When online, certain predetermined guest events like write-faults and code loading, will trigger a trap into the host and give the system the opportunity to verify new instructions.

The new system is guest-transparent; no modifications to the guest operating system are required. However, the whitelist construction is dependent on the guest kernel. Each guest kernel may patch itself in different ways. Correspondingly, the whitelist creation procedure requires deep knowledge of the guest kernel to collect all of the possible valid patches. It has been discovered that the Linux kernel has at least five different facilities that influence code modification, with details presented in [1].

3.2 Kernel Memory Mapping for Rootkit Profiling

The second task of this research is to develop advanced methods and techniques for profiling the behaviors of kernel rootkits. Experience in this research indicates that a major challenge in kernel rootkit profiling is to accurately and completely identify kernel objects that are dynamically created and destructed in kernel memory. In fact, dynamic kernel memory has been a popular target of recent kernel malware due to the difficulty of determining the status of volatile dynamic kernel objects. Some existing approaches use kernel memory mapping to identify dynamic kernel objects and check kernel integrity. The snapshot-based memory maps generated by these approaches are based on the kernel memory which may have been manipulated by kernel malware. In addition, because the snapshot only reflects the memory status at a single time instance, its usage is limited in temporal kernel execution analysis.

In this research, a new runtime kernel memory mapping scheme called allocation-driven mapping is introduced, which systematically identifies dynamic kernel objects, including their types and lifetimes. The scheme works by capturing kernel object allocation and de-allocation events. The system provides a number of unique benefits to kernel malware profiling: (1) an un-tampered view wherein the mapping of kernel data is unaffected by the manipulation of kernel memory and (2) a temporal view of kernel objects to be used in temporal analysis of kernel execution. The effectiveness of allocation-driven mapping is demonstrated in two usage scenarios. First, a hidden kernel object detector can be built that uses an un-tampered view to detect the data hiding attacks of 10 kernel rootkits that directly manipulate kernel objects

(DKOM). Second, a temporal rootkit behavior monitor is developed that tracks and visualizes rootkit behavior triggered by the manipulation of dynamic kernel objects. Allocation-driven mapping enables a reliable analysis of such behavior by guiding the inspection only to the events relevant to the attack.

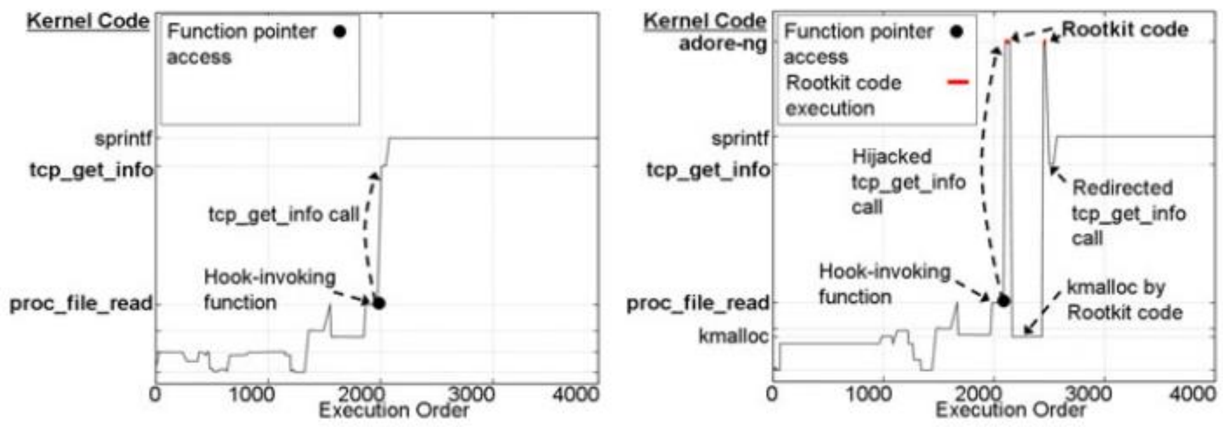


Figure 2: Illustration of adore-ng rootkit KOH behavior (left: before hijacking; right: after hijacking)

The adore-ng attack hijacks kernel code execution by modifying a function pointer and this attack is referred to as Kernel Object Hooking (KOH). This behavior is observed when the influence of a manipulated function pointer is inspected. The system generates two kernel control flow graphs at these samples, each for a period of 4000 instructions. Figure 2 presents how this manipulated function pointer affects runtime kernel behavior. The Y axis presents kernel code; thus, the fluctuating graphs show various code executed at the corresponding time of X axis. A hook-invoking function (proc_file_read) reads the function pointer and calls the hook code pointed to by it. Before the rootkit attack, the control flow jumps to a legitimate kernel function tcp_get_info which calls sprint after that as shown in Figure 2 (left). However, after the hook is hijacked, the control flow is redirected to the rootkit code which calls kmalloc to allocate its own memory, then comes back to the original function (Figure 2 – right).

3.3 Kernel Data Access Behavior Profiling

Based on the kernel memory mapping method presented in Section 3.2, more advanced kernel rootkit behavior models can be defined, which can lead to generic, high-accuracy behavior rootkit signatures. Characterizing malware behavior using its control flow faces several challenges, such as obfuscations in static analysis and the behavior variations in dynamic analysis. This research introduces a new approach to characterizing kernel malware's behavior by using kernel data access patterns unique to the malware. The approach neither uses malware's control flow consisting of temporal ordering of malware code execution, nor the code-specific information about the malware. Thus, the malware signature based on such data access patterns is resilient in matching malware variants.

To evaluate the effectiveness of this approach, the signatures of three classic rootkits are first generated using their data access patterns, and then matched them with a group of kernel execution instances which are benign or compromised by 16 kernel rootkits. The malware signatures did not trigger any false positives in benign kernel runs; however, kernel runs compromised by 16 rootkits were detected due to the data access patterns shared with the compared signature(s). It is further observed that similar data access patterns in the signatures of the tested rootkits and exposed popular rootkit attack operations by ranking common data behavior across rootkits. Experiments show that this new approach is effective not only to detect the malware whose signature is available, but also to determine its variants which share kernel data access patterns.

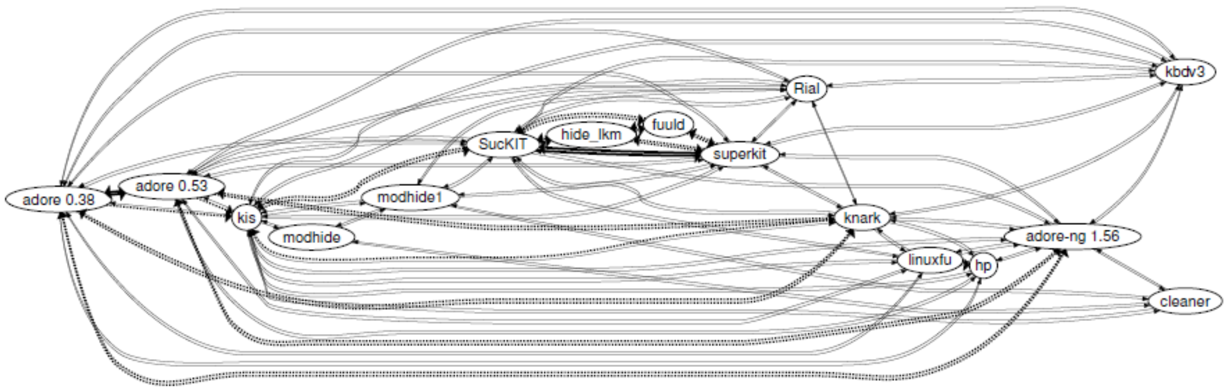


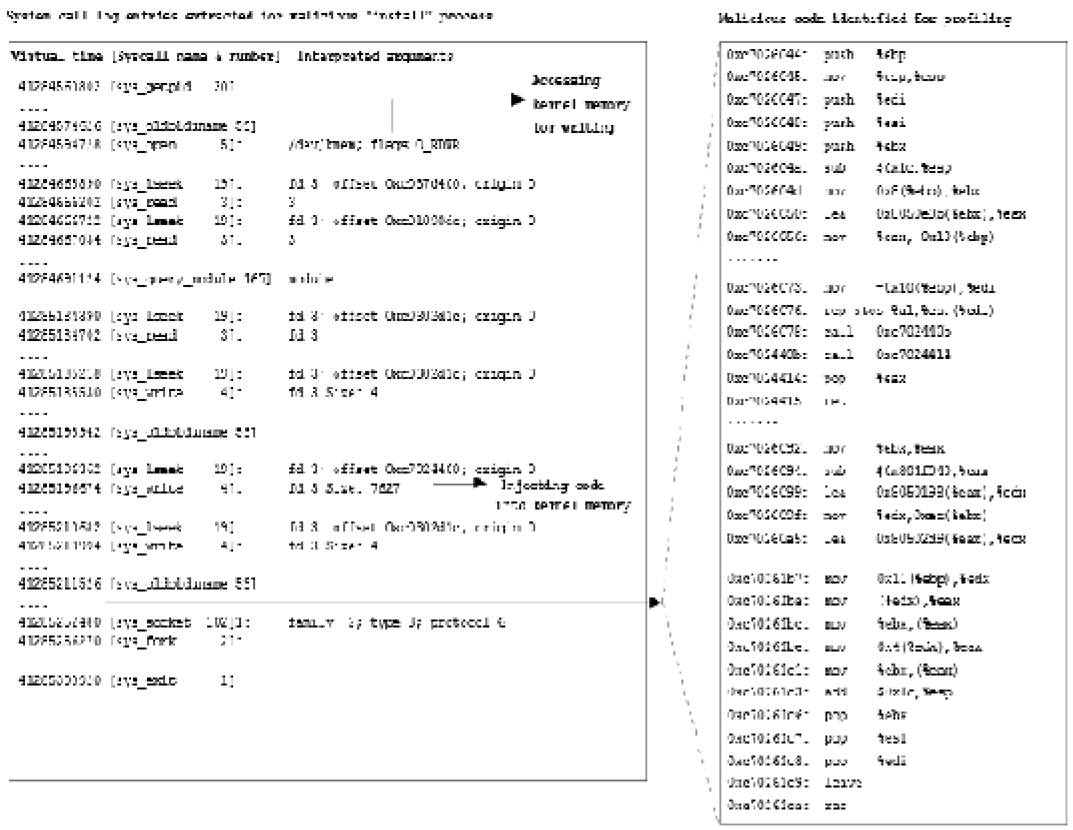
Figure 3: Similarity of data access behavior across rootkits

The similarities of data behavior across rootkits are visualized in Figure 3. A node represents a rootkit signature and an arrow shows the similarity between two signatures using three different arrow types. An arrow from a node M1 to a node M2 means that the signature M1 can be used to determine the rootkit of the signature M2. This figure illustrates that several groups of rootkits have strong similarities. The family of adore rootkits (i.e., adore 0.38, adore 0.53, and adore-ng 1.56) are strongly related in general. The adore-ng 1.56 is connected to other versions with less strong connections, thick dashed arrows, because in newer adore versions (bigger than 1.0 whose name is changed to adore-ng), the internal attack vector is substantially changed to use dynamic objects instead of static objects. A group of rootkits using the /dev/kmem memory device (i.e., SucKIT, hide_lkm, fuuld, and superkit) have a strong relationship to one another. The SucKIT and the superkit are especially connected by using thick solid arrows because they share a majority of data behavior. Some rootkits have relationships with different kinds of rootkits. For example, the kis rootkit is connected to driver-based rootkits such as the adore rootkits and the knark rootkit; but, it is also closely related to /dev/kmem based rootkits such as the SucKIT.

As seen in Figure 3, data access behaviors are not only common in the family of rootkits or similar kinds, but also are present across different kinds of rootkits. The signatures of these related rootkits can be interchangeably used to detect one another.

3.4 Virtualization-based Attack Replay

The kernel malware profiling techniques can be further enhanced to support *live* kernel rootkit monitoring and forensics, especially in the virtual honeypot record and replay (R&R) scenarios. To “rewind” a honeypot’s execution, an intuitive network-level approach would be to replay the captured network traffic targeting the honeypot system (since the honeypot is remotely compromised). However, due to inherent sources of non-determinism in modern systems and software, by simply replaying the captured network packets, it may not be possible to obtain the same execution of the honeypot system. From another perspective, a number of system-level deterministic R&R approaches have been proposed for a variety of purposes, including fault tolerance, application debugging and security analysis. Recording and replaying a VM is well-suited for honeypots since it can capture and reproduce the entire system's execution. However, most prior VM R&R systems are not suitable for high-interaction honeypots because either they do not support commodity OSes or require extensive OS-level customization, or they heavily rely on proprietary virtual machine monitors (VMMs). Moreover, there is a lack of honeypot-specific forensic analysis modules that can take advantage of VM R&R capability.


Accessing kernel memory for writing
Injecting code into kernel memory

System call log entries extracted for malicious "install" process

```
Virtual time [Syscall name & number]  Interpreted arguments
 41284551807 [sys_getpid   20]
 ....
 41284574516 [sys_olduname 59]
 41284594718 [sys_open      5]:   /dev/kmem; flags O_RDWR
 ....
 41284655830 [sys_lseek    19]:   fd 3: offset 0xc0370400; origin 0
 41284656203 [sys_read      3]:   3
 41284656712 [sys_lseek    19]:   fd 3: offset 0xc031004c; origin 0
 41284657044 [sys_read      3]:   3
 ....
 41284691114 [sys_query_module 167]  module

 41285131890 [sys_lseek    19]:   fd 3: offset 0xc0303d1c; origin 0
 41285134702 [sys_read      3]:   fd 3
 ....
 41285135210 [sys_lseek    19]:   fd 3: offset 0xc0303d1c; origin 0
 41285135510 [sys_write     4]:   fd 3 Size: 4
 ....
 41285155342 [sys_olduname 59]
 ....
 41285156302 [sys_lseek    19]:   fd 3: offset 0xc7024400; origin 0
 41285156574 [sys_write     4]:   fd 3 Size: 7627
 ....
 41285211582 [sys_lseek    19]    fd 3: offset 0xc0303d1c; origin 0
 41285211994 [sys_write     4]:   fd 3 Size: 4
 ....
 41285211516 [sys_olduname 59]
 ....
 41285252480 [sys_socket  102(1)]:   family 2; type 3; protocol 6
 41285256710 [sys_fork      2]:

 41285300010 [sys_exit      1]
```

Malicious code identified for profiling

```
0xc7026c44:  push   %ebp
0xc7026c45:  mov    %esp,%ebp
0xc7026c47:  push   %edi
0xc7026c48:  push   %esi
0xc7026c49:  push   %ebx
0xc7026c4a:  sub    $0x1c,%esp
0xc7026c4d:  mov    0x8(%ebp),%edx
0xc7026c50:  lea    0x1000e00(%ebx),%eax
0xc7026c56:  mov    %eax, 0x10(%ebp)
.......

0xc7026c73:  mov    -0x10(%ebp),%edi
0xc7026c76:  rep stos %al,%es:(%edi)
0xc7026c78:  call   0xc702440b
0xc702440b:  call   0xc7024411
0xc7024414:  pop    %eax
0xc7024415:  ret
.......

0xc7026c92:  mov    %ebx,%eax
0xc7026c94:  sub    $(0x801f040),%eax
0xc7026c99:  lea    0x8050139(%eax),%ecx
0xc7026c9f:  mov    %edx,0xac(%ebx)
0xc7026ca5:  lea    0x8050239(%eax),%ecx

0xc70261b7:  mov    0x1c(%ebp),%edx
0xc70261ba:  mov    (%edx),%eax
0xc70261bc:  mov    %ebx,(%eax)
0xc70261be:  mov    0x4(%edx),%eax
0xc70261c1:  mov    %ebx,(%eax)
0xc70261c3:  add    $0x1c,%esp
0xc70261c6:  pop    %ebx
0xc70261c7:  pop    %esi
0xc70261c8:  pop    %edi
0xc70261c9:  leave
0xc70261ca:  ret
```

Figure 4: Log excerpt for live forensic analysis of SuKIT rootkit using Timescope

In this research, a VM R&R system called Timescope is developed. Timescope is a time-traveling high-interaction honeypot system designed for extensible, fine-grained forensic analysis. Leveraging previous insights from VM-level R&R systems, an open-source tool is developed, aiming to engage the security community and benefit related research efforts that may require similar features. In addition, the system is extended by developing a number of honeypot-specific analysis modules: contamination graph generator (I), transient evidence recoverer (II), shellcode extractor (II), and break-in reconstructor (IV). These modules are applied only during honeypot execution replay sessions and placed externally so that the replay itself is not perturbed. By allowing the analysis modules to "travel back in time", it addresses key questions in honeypot forensic investigations, such as: "what are the contaminations or damages caused by an intrusion?"; "what intermediate evidence (e.g., files and directories), if any, has been erased by the attacker?"; "how is the attack launched?"

Timescope and these analysis modules have been implemented based on the open-source QEMU VMM and enabled multi-faceted, inter-related malware forensic analysis during multiple replay sessions. Evaluation with a number of attack scenarios, including real-world worm programs and kernel rootkits, shows the practicality and effectiveness of Timescope to repeatedly and comprehensively analyze past intrusions. The experiments are enabled by repeatedly rewinding the honeypot's execution, not based on the log from one single run.

In one of the experiments, the goal is to demonstrate how Timescope's replay-based forensic analysis techniques can be used to analyze intermediate memory states in the honeypot. For this, the SucKIT kernel rootkit is launched to attack a honeypot VM. To analyze this attack, a replay session is run with the first analysis module and the root login and subsequent commands executed (with the sys_execve() system call) are observed. A subset of the log is shown in Figure 4. In particular, the command "install" is run by the attacker and it opens the file /dev/kmem which, gives complete write access to the root user to write to arbitrary locations in the kernel memory. To highlight a subset of the execution profiling analysis, consider the lines indicating that the kernel memory is being overwritten as shown in Figure 4. These lines indicate kernel memory being overwritten from the ranges 0xc7024400 to 0xc70261cb. One interesting observation from Figure 4 is that the "install" user process is issuing a sys_oldolduname() system call, when in reality, the rootkit overwrote the address of this system call handler in the kernel multiple times to use it for allocating kernel memory, injecting rootkit code in the kernel space, and hijacking kernel control flow.

3.5 Process Implanting for Agile Assured Execution

This research has also led to two interesting (and novel) capabilities for agile and assured process execution in VMs: process implanting and process out-grafting. This section describes process implanting and the next section will present process out-grafting.

To motivate process implanting, it has been known that many security tools for malware detection are vulnerable to attacks because they are exposed to malwares. The most common technique used by a malware to hide itself is to dysfunction anti-malware engines. In order to be tamper-resistant and stealthy, current approaches leveraging virtualization technology to detect and prevent malware attacks usually try to move the monitoring tools from the untrusted guest VM to underlying hypervisor or to another isolated trusted VM to prevent the tools from being tampered with. By reconstructing the semantic view of guest VM from host through the technique of virtual machine introspection, a large (yet incomplete) amount of semantic information can be obtained. The semantic gap between the guest OS and host is a known barrier to services operating below the abstractions of guest OS and applications. This problem becomes more challenging with the widely deployment of hardware virtualization, whose goal is to run most of the native instructions directly on the CPU and expose as few details as possible to the VMM to gain higher performance. Previous approaches to virtual machine introspection can passively detect or actively monitor attacks with effectiveness. Yet it is desirable to act even more actively in the counterattacks by obstructing, analyzing and subverting the malware attacks. This requires a more complete, native semantic view of the guest VM.

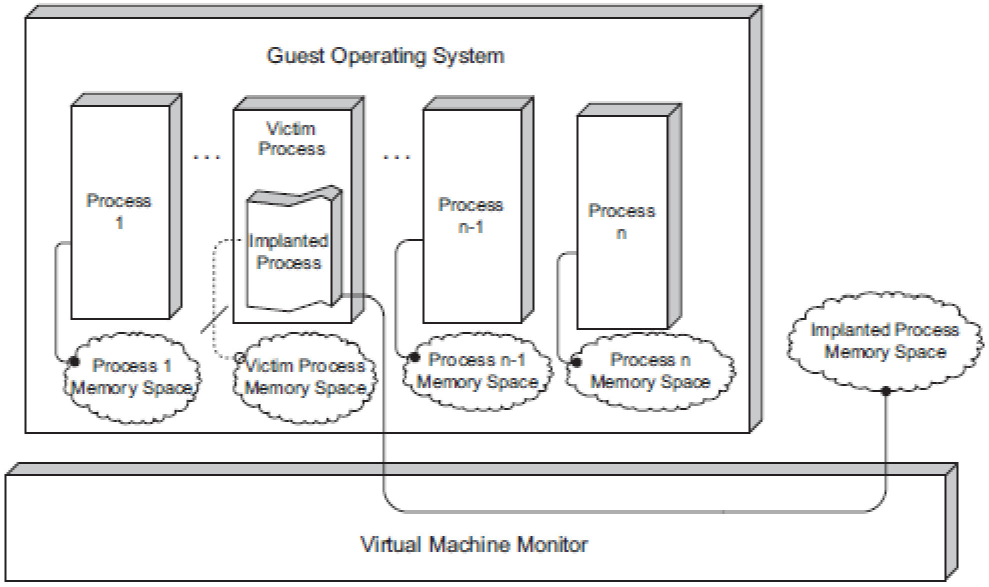
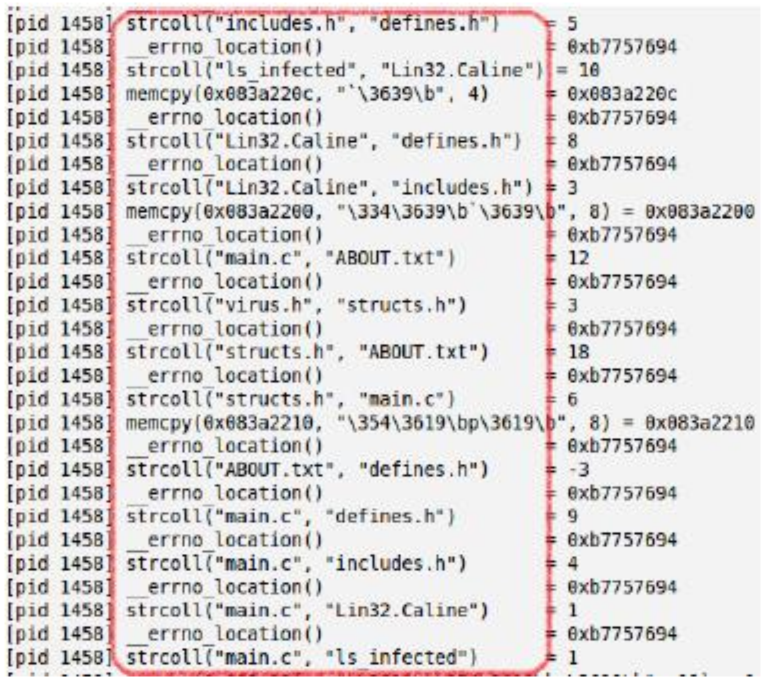


Figure 5: Overview of process implanting

This research has resulted in process implanting, a general-purpose active VM introspection framework. The idea is to implant a process directly from the host into the guest under the cover of an existing process inside the guest OS to narrow the semantic gap and gain in-context knowledge of the running VM. Instead of leaving the implanted process alone inside the guest VM, a series of coordination and protection mechanisms are designed. They are supported by the higher privileged hypervisor to exempt the implanted process from malware's tampering and leave minimum negative impact on the normal execution of guest OS and applications after it exits.



```
[pid 1458] strcoll("includes.h", "defines.h")           = 5
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("ls_infected", "Lin32.Caline")       = 10
[pid 1458] memcpy(0x083a220c, "`\3639\b", 4)            = 0x083a220c
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("Lin32.Caline", "defines.h")         = 8
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("Lin32.Caline", "includes.h")        = 3
[pid 1458] memcpy(0x083a2200, "\334\3639\b`\3639\b", 8) = 0x083a2200
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("main.c", "ABOUT.txt")               = 12
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("virus.h", "structs.h")              = 3
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("structs.h", "ABOUT.txt")            = 18
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("structs.h", "main.c")               = 6
[pid 1458] memcpy(0x083a2210, "\354\3619\bp\3619\b", 8) = 0x083a2210
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("ABOUT.txt", "defines.h")            = -3
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("main.c", "defines.h")               = 9
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("main.c", "includes.h")              = 4
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("main.c", "Lin32.Caline")            = 1
[pid 1458] __errno_location()                           = 0xb7757694
[pid 1458] strcoll("main.c", "ls_infected")             = 1
```

Figure 6: Log excerpt from process implanting-enabled library call introspection

In one of the experiments, the ltrace tool is implanted to trace the ls utility which had been infected by caline. Caline is an ELF infector using S.P.I (segment padding infection) technique. It inserts virus code after the code segment of an ELF binary to change its behavior. Through tracing the infected ls, it becomes easy to identify the deviated execution path by checking the arguments of library calls. The box in Figure 6 presents the suspicious execution results.

3.6 Process Out-grafting for Agile Assured Execution

As a counter-part of process implanting (Section 3.5), process out-grafting is an architectural approach that addresses both isolation and compatibility challenges for out-of-VM, fine-grained user-mode process execution monitoring. Similar to prior out-of-VM approaches, out-grafting still confines vulnerable systems as VMs and deploys security tools outside the VMs. However, instead of analyzing the entire VM on all running processes, out-grafting focuses on each individual process for fine-grained execution monitoring. More importantly, this approach is designed to naturally support existing user-mode process monitoring tools (e.g., strace, ltrace, and gdb) outside of monitored VMs on an internal suspect process, without the need of modifying these tools or making them introspection-aware (as required in prior out-of-VM approaches). For simplicity, the terms “production VM” and “security VM” are used respectively to represent the vulnerable VM that contains a suspect process and the analysis VM that hosts the security tool to monitor the suspect process.

To enable process out-grafting, two key techniques are developed (shown in Figure 7). Specifically, the first technique, on-demand grafting, relocates the suspect process on demand from the production VM to security VM (that contains the process monitoring tool as well as its supporting environment). By doing so, grafting effectively brings the suspect process to the monitor for fine-grained monitoring, which leads to at least two important benefits: (1) By co-locating the suspect process to run side-by-side with the monitor, the semantic gap caused by the VM isolation is effectively removed. In fact, from the monitor's perspective, it runs together with the suspect process inside the same system and based on its design can naturally monitor the suspect process without any modification. (2) In addition, the monitor can directly intercept or analyze the process execution even at the granularity of user-level function calls, without requiring hypervisor intervention, which has significant performance gains from existing introspection-based approaches.

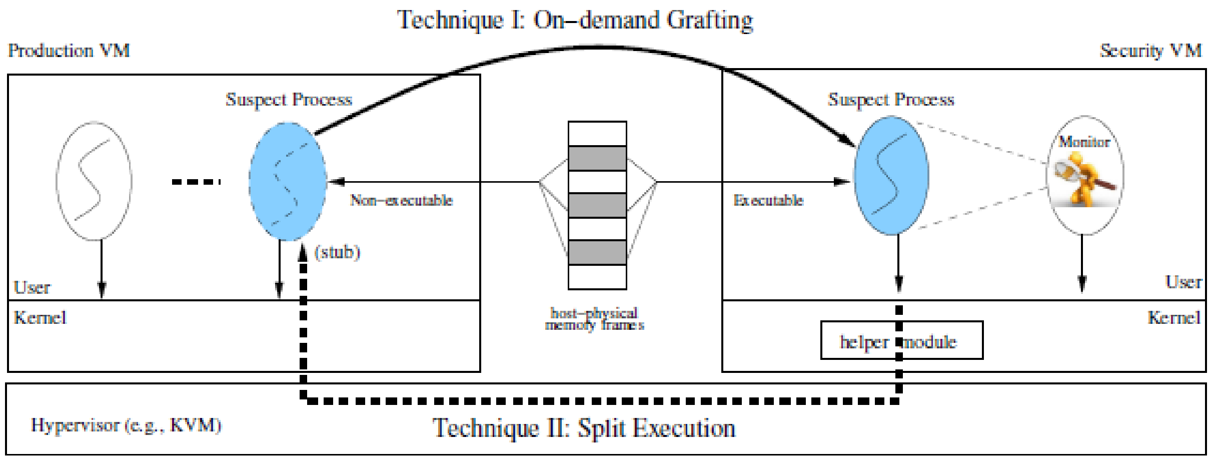


Figure 7: Overview of process out-grafting

To effectively confine the (relocated) suspect process, the second technique enforces a mode-sensitive split execution of the process, thus the name split execution. Specifically, only the user-mode instructions of the suspect process, the main focus for fine-grained monitoring, will be allowed to execute in the security VM; all kernel-mode execution that requires the use of OS kernel system services is forwarded back to the production VM. By doing so, the system can not only maintain a smooth continued execution of the suspect process after relocation, but ensure its isolation from the monitoring tools. Particularly, from the suspect process' perspective, it is still logically running inside the production VM. In the meantime, as the suspect process physically runs inside the security VM, the monitoring overhead will not be inflicted to the production VM, thus effectively localizing monitoring impact within the security VM.

A proof-of-concept prototype has been implemented on KVM/ Linux (version kvm-2.6.36.1) and tested to out-graft various processes from different VMs running either Fedora 10 or Ubuntu 9.04. It has been evaluated with a number of different scenarios, including the use of traditional process monitoring tools, i.e., strace/ltrace/gdb, to monitor an out-grafted process from another VM. Note that these fine-grained process monitoring tools cannot be natively supported if the semantic gap is not effectively removed. Moreover, it is also shown that advanced (hardware-assisted) monitoring tools can be deployed in the security VM to monitor a process in the production VM, while they may be inconvenient or even impossible to run inside the production VM. The performance evaluation with a number of standard benchmark programs shows that the process out-grafting prototype incurs a small performance overhead and the monitoring overhead is largely confined within the security VM, not the production VM.

In one of the experiments, process out-grafting is applied to fight malware obfuscation. Most recent malware apply obfuscation techniques to evade existing malware detection tools. Code packing is one of the popular obfuscation techniques. To detect packed code, efficient behavioral monitoring techniques such as OmniUnpack have been developed to perform real-time monitoring of a process' behavior by tracking the pages it writes to and then executes from. When the process invokes a "dangerous" system call, OmniUnpack looks up its page list to determine whether any previously written page has been executed from. If so, this indicates packing behavior, at which point a signature-based anti-virus tool can be invoked to check the process' memory for known malware.

In the experiment, a freely available UPX packer is used to pack the Kaiten bot binary. A security-sensitive event trigger is also utilized to initiate process out-grafting when a suspect process invokes the sys_execve system call. The trigger is placed such that just before the system call returns to user-mode (to execute the first instruction of the new code), KVM is invoked to out-graft the process' execution to the security VM. Inside the security VM, the OmniUnpack tool is running to keep track of page accesses by the process. Since the system calls invoked by the process are also available for monitoring, OmniUnpack successfully detects the packing behavior.

3.7 Reducing Kernel Attack Surface via Face-Change

This research also explores the paradigm of treating the guest OS as a "moving target" for better protection against kernel-level (as well as user-level) attacks. Modern operating systems strive to shrink the size of the trusted computing base (TCB) to ease code verification and minimize trust assumptions. For a general-purpose OS like Linux, kernel minimization has already been established as a practical approach to reducing attack surface. But for a general-purpose OS supporting a variety of applications, whole-system profiling unnecessarily enlarges the kernel attack surface of the system.

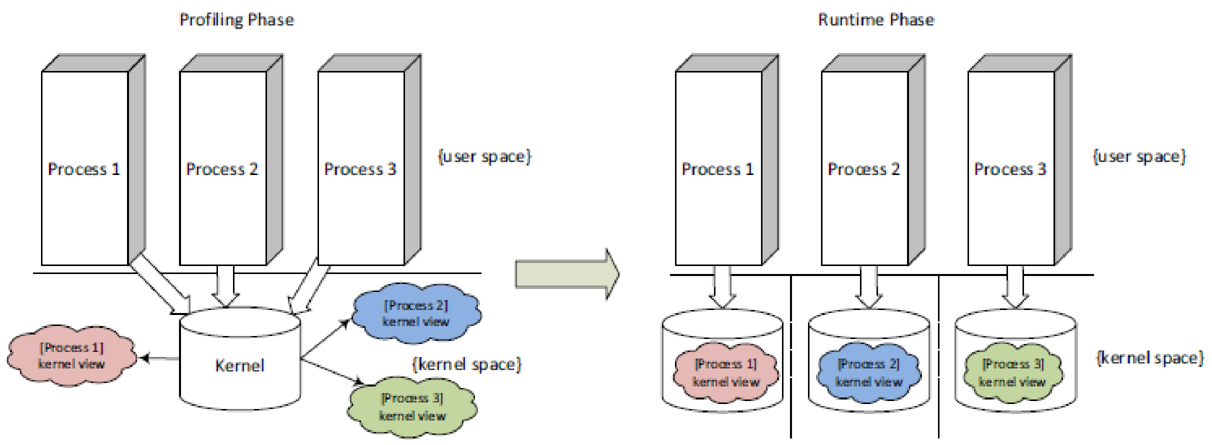
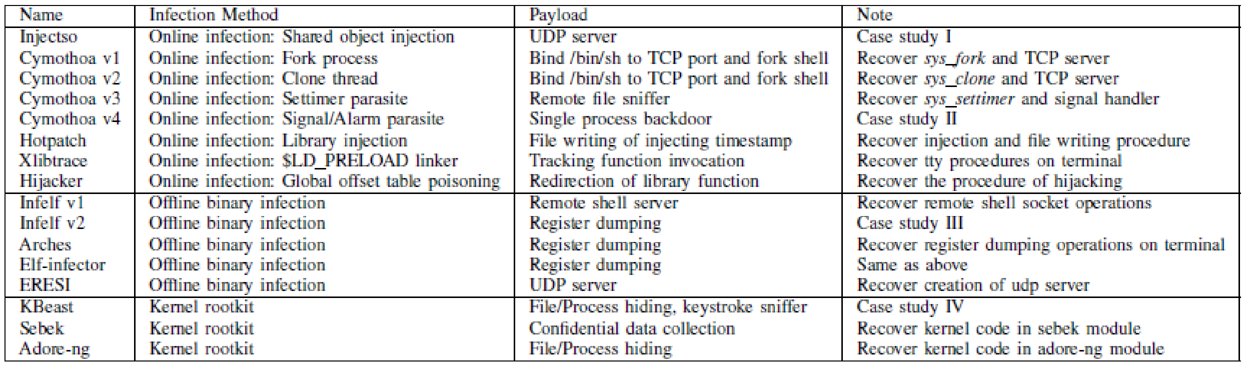


Figure 8: Overview of Face-Change where processes each have their own kernel code views

A key observation from this research is that kernel code executed under different application contexts varies drastically. Profiling experiments show that two distinct applications may share as little as 33.6% of their executed kernel code – thus system-wide kernel minimization would over-approximate both applications' kernel requirements. For example, the kernel functionality needed by task manager *top* is to read statistics data from the memory-based *proc* file system and write to the tty device. In sharp contrast, the *Apache* web server primarily requires network I/O services from the kernel. If a system running *top* and *Apache* simultaneously is profiled, the kernel's networking code will be exposed to *top* simply because *Apache* is in the same VM. Further, assume *top* is the target of a malicious attack, the compromised *top* may be implanted with a parasite network server as a backdoor without violating the minimized kernel's constraint.

To address the problems with whole-system-based kernel minimization, FACE-CHANGE is developed, which is a virtualization-based system to support *dynamic switching among multiple minimized kernels, each for an individual application*. The term *kernel view* refers to the in-memory kernel code presented to an individual application. In conventional kernels, all concurrently running user-level processes share the same kernel view containing the entire kernel code section, which is referred to as a *full kernel view*. FACE-CHANGE aims to present each process with a different, customized kernel view, which is prepared individually in advance by profiling the application's needs. Any unnecessary kernel code is eliminated to minimize the attack surface accessible to this specific application. At runtime, FACE-CHANGE identifies the current process context and dynamically switches to its customized kernel view.



Name	Infection Method	Payload	Note
Injectso	Online infection: Shared object injection	UDP server	Case study I
Cymothoa v1	Online infection: Fork process	Bind /bin/sh to TCP port and fork shell	Recover *sys_fork* and TCP server
Cymothoa v2	Online infection: Clone thread	Bind /bin/sh to TCP port and fork shell	Recover *sys_clone* and TCP server
Cymothoa v3	Online infection: Settimer parasite	Remote file sniffer	Recover *sys_settimer* and signal handler
Cymothoa v4	Online infection: Signal/Alarm parasite	Single process backdoor	Case study II
Hotpatch	Online infection: Library injection	File writing of injecting timestamp	Recover injection and file writing procedure
Xlibtrace	Online infection: $LD_PRELOAD linker	Tracking function invocation	Recover tty procedures on terminal
Hijacker	Online infection: Global offset table poisoning	Redirection of library function	Recover the procedure of hijacking
Infelf v1	Offline binary infection	Remote shell server	Recover remote shell socket operations
Infelf v2	Offline binary infection	Register dumping	Case study III
Arches	Offline binary infection	Register dumping	Recover register dumping operations on terminal
Elf-infector	Offline binary infection	Register dumping	Same as above
ERESI	Offline binary infection	UDP server	Recover creation of udp server
KBeast	Kernel rootkit	File/Process hiding, keystroke sniffer	Case study IV
Sebek	Kernel rootkit	Confidential data collection	Recover kernel code in sebek module
Adore-ng	Kernel rootkit	File/Process hiding	Recover kernel code in adore-ng module

Figure 9: Evaluation results showing FACE-CHANGE detecting both kernel and user-level malware

Extensive experiments have been performed to evaluate the effectiveness of FACE-CHANGE with 13 user-level malware (8 of them use online runtime infection and 5 use offline binary infection) and 3 kernel-level rootkits. The results are presented in Figure 9.

3.8 Compiler Support for Kernel Rootkit Defense via Randomization

Finally, to complement the virtualization-based kernel protection techniques, a compiler-based solution has been proposed that does not require virtualization support. Although this research focuses on the virtualization-based approaches, a non-virtualized solution is helpful for comparison and complement.

The vast majority of hosts on the Internet, including mobile clients, are running on one of three major operating system families. Malicious operating system kernel software, such as the code introduced by a kernel rootkit, is strongly dependent on the organization of the victim operating system. Due to the lack of diversity of operating systems, attackers can craft a single kernel exploit that has the potential to infect millions of hosts. If the underlying structure of vulnerable operating system components has been changed, in an unpredictable manner, then attackers must create many unique variations of their exploit to attack vulnerable systems en masse. If enough variants of the vulnerable software exist, then mass exploitation is much more difficult to achieve. Many forms of automatic software diversification have been explored and found to be useful for preventing malware infection. Forrest et. al. make a strong case for software diversity and describe a few possible techniques including: adding or removing nonfunctional code, reordering code, and reordering memory layouts. The techniques in this research build on the latter.

Two different ways are proposed to mutate an operating system kernel using memory layout randomization to resist kernel-based attacks. A new method is introduced for randomizing the stack layout of function arguments. Additionally, a previous technique for record layout randomization is refined by introducing a static analysis technique for determining the randomizability of a record. Prototypes of these techniques are developed using the plugin architecture offered by GCC. To test the security benefits of the techniques, multiple Linux kernels have been randomized using the new compiler plugins. The randomized kernels are then attacked using multiple kernel rootkits. Experiments show that by strategically selecting just a few components for randomization, the techniques prevent all kernel rootkits in the experiments.

4. Publications

[1] Dannie Stanley, Zhui Deng, Dongyan Xu, Rick Porter, Shane Snyder, “Guest-transparent instruction authentication for self-patching kernels”, Proceedings of IEEE Conference on Military Communications (MILCOM 2012), 2012.

[2] Junghwan Rhee, Ryan Riley, Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, "Data-Centric OS Kernel Malware Characterization", IEEE Transactions on Information Forensics and Security (TIFS), 9(1), 2014.

[3] Junghwan Rhee, Zhiqiang Lin, Dongyan Xu, "Characterizing Kernel Malware Behavior with Kernel Data Access Patterns," Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2011), 2011.

[4] Deepa Srinivasan, Xuxian Jiang, "Time-traveling Forensic Analysis of VM-based High-interaction Honeypots," Proceedings of the 7th International Conference on Security and Privacy in Communication Networks (SecureComm 2011), 2011.

[5] Zhongshu Gu, Zhui Deng, Dongyan Xu, Xuxian Jiang, "Process Implanting: A New Active Introspection Framework for Virtualization," Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS 2011), 2011.

[6] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, Dongyan Xu, "Process Out-Grafting: An Efficient 'Out-of-VM' Approach for Fine-Grained Process Execution Monitoring," Proceedings of the ACM Conference on Computer and Communications Security (CCS 2011), 2011.

[7] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, Dongyan Xu, "FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine," Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014), 2014.

[8] Dannie Stanley, Dongyan Xu, Eugene H. Spafford, “Improved Kernel Security Through Memory Layout Randomization”, Proceedings of IEEE International Performance, Computing, and Communications Conference (IPCCC 2013), 2013.